YARD Architecture Reference

Draft version August 18, 2011

Contents

ααα Shorthand marginal notes indicating the status of a given feature can be decoded as follows: ***ααα***
"**??**" indicates ill-defined
"**%%**" indicates partial implementation
"**$$**" indicates an optional feature

1 Overview

YARD = Yet Another RISC Design

The YARD-1 is a simple 32 bit[1] RISC architecture having 16 registers and using a compact 16 bit, two operand instruction format.

A synthesizable BSD-licensed VHDL core, the Y1A, provides a rudimentary implementation of the YARD-1, and is intended for embedded FPGA designs utilizing internal block memory for code and data storage.

A portable[2] no-frills absolute cross assembler is provided with the Y1A core, along with an instruction set verification testbench.

2 Programming Model

2.1 Register File

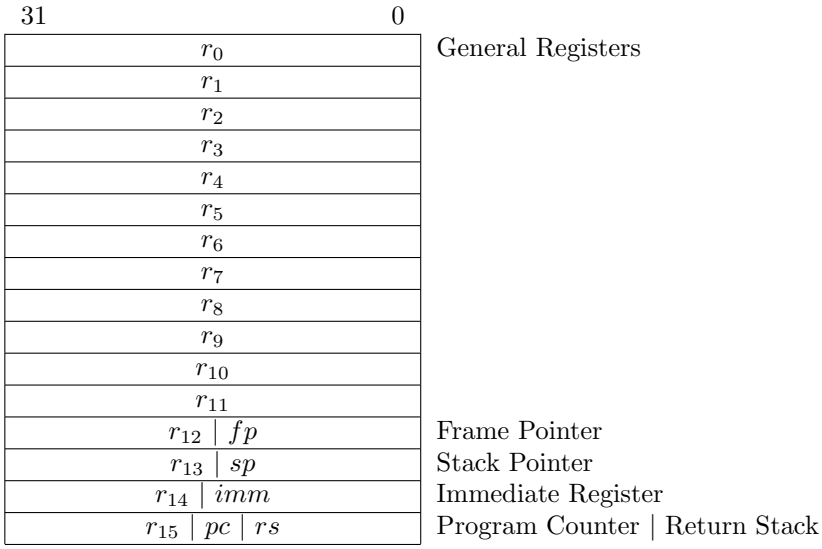

Figure 1: Register File

The first twelve registers, $r_0 \ldots r_{11}$, are completely general purpose.

Registers r_{12} and r_{13}, aka fp and sp, are available as base registers for the stack offset addressing mode, but are otherwise undedicated.

Register r_{14}, aka imm, is loadable via the `imm12` and `ldi` opcodes to provide arbitrary immediate values (see section 4); this register also sources the offset field for offset indirect addressing. Otherwise, r_{14} may be used and referenced like any other general purpose register.

Register r_{15} is a dedicated register which provides access to either the current program counter or the hardware return stack:

- When used as the base register in address calculations (`ld, st, lea`), references to r_{15} provide the PC value of the current instruction.
- When accessed as a `mov/ld/st` data operand, references to r_{15} push or pop the top of the internal hardware return stack.

[1] Early versions had an optional 16 bit datapath configuration, which is not currently supported.
[2] Cross assembler written in Perl.

2.2 Special Registers

?? Special registers, for processor status and control, will be eventually accessible through the system coprocessor. Likely register definitions include:

31 … 0	
status	Status Register
control	Control Register
tick_msw	Clock Ticks (MSW)
tick_lsw	Clock Ticks (LSW)
type	Processor Type
options	Processor Options (optional instuctions, stack depth)
id_msw	ID (MSW)
id_lsw	ID (LSW)

Figure 2: Special Registers

2.3 Return Stack

A hardware return stack is used to store return addresses and processor state for subroutine calls and interrupts; stack depth is implementation specific, with a minimum depth of 16.

%% The top of this return stack is accessible via `mov/ld/st` data operations on $r_{15} \mid rs$, allowing return addresses for non-leaf functions to be moved onto a software stack for unlimited call depth.

2.4 Register Calling Conventions

?? Preliminary, subject to change upon the existence of a compiler.

31 … 0	
r_0	Function Return Value
r_1	Parameter
r_2	Parameter
r_3	Parameter
r_4	Parameter
r_5	Callee Register
r_6	Callee Register
r_7	Callee Register
r_8	Callee Register
r_9	Caller Register (save if used)
r_{10}	Caller Register (save if used)
r_{11}	Caller Register (save if used)
$r_{12} \mid fp$	Caller Register (save if used) | Frame Pointer
$r_{13} \mid sp$	Stack Pointer
$r_{14} \mid imm$	Immediate Register, used by compiler and assembler to build arbitrary constants

Figure 3: Calling Convention

2.5 Input Flags

The processor includes 16 hardware input flags, which can be individually tested using the skip-on-flag instructions (`skip.fs, skip.fc`).

2.6 Coprocessor Interface

?? The coprocessor interface has not been fully defined; one opcode has been reserved for coprocessor operations, along with a skip slot for coprocessor conditionals.

Provision exists for up to 16 coprocessors, with the first eight intended for system and standardized usage (control, memory, math, etc.), and the last eight available for user extensions.

The coprocessors have read visibility into the current register file state, allowing coprocessor calls to resemble normal function calls.

system	0 : System Control
mmu	1 : Memory Management Unit
–	2 : reserved
–	3 : reserved
imath	4 : Integer Math
–	5 : reserved
fpu	6 : Floating Point Unit
–	7 : reserved
user	8 : User defined
user	9 : User defined
user	10 : User defined
user	11 : User defined
user	12 : User defined
user	13 : User defined
user	14 : User defined
user	15 : User defined

Figure 4: Coprocessors

3 ALU Operations

ALU operations have the general form:

$r_a \leftarrow r_a \;\; OP \;\; op_b$ written as $OP \;\; r_a, \;\; op_b$ in assembly code

Where r_a is a register, and op_b is either a register r_b, or a short immediate constant (the short encodings accomodate 5 bit signed, 2^N, and $2^N - 1$ constants).

Selectable inversion of the B operand is provided for the logical instructions, allowing for convenient bit mask generation when used with the 2^N and $2^N - 1$ encoded short immediates.

mnemonic	*name*
`mov{.not}`	MOVe
`and{.not}`	AND
`or{.not}`	OR
`xor{.not}`	eXclusive OR
`add`	ADD
`sub`	SUBtract
`rsub`	Reverse SUBtract
`lsr`	Logical Shift Right
`lsl`	Logical Shift Left
`asr`	Arithmetical Shift Right
`ror`	ROtate Right
`rol`	ROtate Left
`flip`	bit FLIP
`ff1`	Find First 1
`cnt1`	CouNT 1's

Table 1: ALU Instructions

Note that in assembly code, the destination register operand r_a comes first:

```
mov r0,r1 ; r0 = r1
add r0,r4 ; r0 = r0 + r4
sub r0,r4 ; r0 = r0 - r4
skip.ge r0,r4 ; r0 >= r4
```

4 Immediates

Provision for immediates beyond the reach of the op_b short encodings is available via r_{14}, the *imm* register. *Imm* may be loaded either via any normal register operation, or by using one of the two dedicated immediate instructions:

- `imm12 #CONSTANT ; IMMediate, 12 bit`
  Loads a 12 bit sign-extended value into *imm*
- `ldi LABEL ; LoaD Immediate`
  Loads a PC Relative quad (0..4095 quad offset) into *imm*

?? A future version of the assembler will support an `imm` macro to load *imm* using the most compact encoding method (encoded short immediate `mov`, `imm12`, or `ldi`).

```
imm #CONSTANT ; IMMediate, select best encoding
```

5 Memory References

5.1 Operand Sizing

Memory references support three operand sizes :

- byte = 8 bits
- wyde[3] = 16 bits
- quad = 32 bits

5.2 Load and Store

Memory is byte addressable, with address alignment **REQUIRED** for 16/32 bit operands.

LD allows sign or zero extended 8/16/32 bit memory reads.

ST allows 8/16/32 bit memory writes.

5.3 Byte Order and Bit Numbering

Byte order in memory is Big-Endian, having the most significant byte of a multi-byte operand stored at the lowest memory address.

Bit numbering in registers, buses, and other multi-bit fields is Little-Bittian, wherein an N bit field has the MSB numbered N-1, and the LSB numbered zero; thus bit number N has a numerical weight of 2^N.

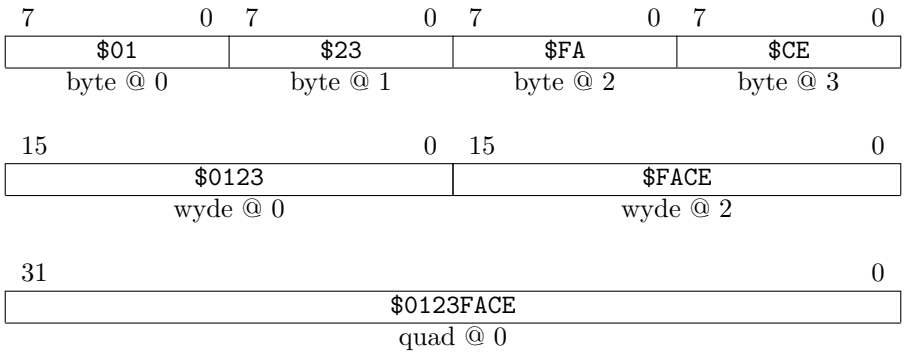

Figure 5: Operands in Memory

[3]per Knuth"MMIXware",1999,p.4: "Weight Watchers know that two nybbles make one byte, but two bytes make one wyde"

5.4 Data Addressing Modes

5.4.1 Fundamental Addressing Modes

- Register Indirect: `(Ra)`

```
ld   r0, (r4)
```

  The effective address is sourced by register r_a .

- Register Offset Indirect: `.imm(Ra)`

```
imm12  #$3E4
ld  r0, .imm(r4)
```

  The effective address is formed from the sum $r_a + imm$.

  A prefix instruction (`imm12` or `ldi`), or other register operation, is needed to first load imm .

?? Once the assembler supports the automatic generation of constant prefixes, the offset will be able to be supplied directly in the memory reference:

```
ld  r0, some_offset(r4)
```

- Stack Offset: `stk_offset(fp | sp)`

```
ld.l  r0, 48(sp)
```

  This addressing mode supports **ONLY** quad loads and stores.

  The *sp* and *fp* registers **MUST** be quad aligned when using this address mode.

  The stack offset is encoded directly in the instruction as a 4 bit field, yielding an offset range of 0-15 quads (in assembler, stk_offset is a byte offset with an allowed range of 0,4 . . . 60 bytes); offsets outside this range should be assembled as a normal register offset indirect access.

5.4.2 Synthetic Addressing Modes

The use of register offset indirect addressing with *pc* and *imm* as the base register allows PC relative and absolute address modes to be synthesized, in both short (`imm12` prefix) and long (`ldi` prefix) variants.

As the `imm12` prefix is signed, 'short absolute' mode can reach the top and bottom 2K bytes of memory, which works well for memory and I/O in tiny embedded systems.

?? The assembler does **NOT** currently support automatic generation of these addressing modes, but when it does the syntax will look something like this:

- PC Relative: `label(pcr)`

```
lea   r0, some_label(pcr)
```

- Absolute: `address`

```
st.b   r0, $ffff_ff00
```

6 Branches & Jumps

6.1 Branch and Jump Addressing

The jump instructions (`jmp/jsr`) use a 32 bit absolute address in a register, e.g. `jmp (r1)`.

The branch instructions support PC relative addressing in both short and long variants, with instruction (16 bit) offsets of 9, 21, and 31 bits:

instruction(s)	*offset size*	*branch range, instructions*
`bra | bsr`	9 bit	$+255 \ldots -256$
`imm12 + lbra | lbsr`	21 bit	$+1,048,575 \ldots -1,048,576$
`ldi + lbra | lbsr`	31 bit	$+2^{30}-1 \ldots -2^{30}$
`rbra | rbsr`	31 bit	$+2^{30}-1 \ldots -2^{30}$

Table 2: Branch Offsets

The long branch variants use an `imm12` or `ldi` prefix to load *imm* with the upper bits of the branch offset, which is combined with 9 bits from the branch instruction field to form the 21 or 31 bit instruction offset.

6.2 Delay Slots

The programming model exposes one delay slot for the change-of-flow instructions; execution of the delay slot instruction is enabled by the `.d` suffixed instruction variants, e.g. `bra.d, jmp.d`

7 Skips

7.1 Skip instruction

Conditional execution is supported via the `skip` instruction, which nullifies the execution of the following instruction if the skip condition is true.

Supported `skip` conditions (see SCC table in section 9.5) include register-register, register-zero, and assorted bit/byte/wyde tests; the `add/sub/rsub` instructions also provide a skip-no-carry/borrow mode.

Conditional change-of-flow is obtained by skipping a branch or jump.

7.2 Multi-instruction skips

%% The `spam` [`Skip Propagate Against Mask`] instruction, when placed in the shadow of a skip, propagates the skip test result to the following eight instructions using either AND or XOR masking modes, whereby any given instruction is skipped if the corresponding bit of the resulting skip vector is set.

`Spam` in AND mode [$skip_n \leftarrow skip\_test\ AND\ skip\_mask_n$] skips the masked instructions when the skip condition is true, and always executes the non-masked instructions, thus extending the basic skip operation to multiple instructions (which may be intertwined with non-skipped instructions).

`Spam` in XOR mode [$skip_n \leftarrow (skip\_test\ XOR\ skip\_mask_n)\ AND\ trunc\_mask_n$)] implements a branch free if-else construct for short instruction sequences, in which only one of the two sets of instructions is executed. $Trunc\_mask$ allows the XOR mode sequence length to be shortened to fewer than eight instructions.

8 Instruction Summary (Alphabetic by Mnemonic)

Opcode	Mnemonic	Name
01000ttbbbbbaaaa	add	ADD
01001ttbbbbbaaaa	add.snc	ADD, Skip No Carry
00010ttbbbbbaaaa	and	AND
00011ttbbbbbaaaa	and.not	AND NOT
0111010bbbbbaaaa	asr	Arithmetical Shift Right
1110000rrrrrrrrr	bra	BRAnch
1110001rrrrrrrrr	bra.d	BRAnch, Delayed
1110010rrrrrrrrr	bsr	Branch SubRoutine
1110011rrrrrrrrr	bsr.d	Branch SubRoutine, Delayed
000000100000aaaa	clr	CLeaR
01111111bbbbaaaa	cnt1	CouNT 1's
0101001000001aaaa	dec	DECrement
01111110bbbbaaaa	ff1	Find First 1
0111011bbbbbaaaa	flip	FLIP
1011iiiiiiiiiiii	imm12	IMMediate, 12 bit
0100001000001aaaa	inc	INCrement
111110000000aaaa	jmp	JuMP
111110100000aaaa	jmp.d	JuMP, Delayed
111111000000aaaa	jsr	Jump SubRoutine
111111100000aaaa	jsr.d	Jump SubRoutine, Delayed
1110100rrrrrrrrr	lbra	Long BRAnch
1110101rrrrrrrrr	lbra.d	Long BRAnch, Delayed
1110110rrrrrrrrr	lbsr	Long Branch SubRoutine
1110111rrrrrrrrr	lbsr.d	Long Branch SubRoutine, Delayed
10000ssmbbbbbaaaa	ld	LoaD
10001ssmbbbbbaaaa	ld.b	LoaD, Byte
10000ssmbbbbbaaaa	ld.q	LoaD, Quad
10001ssmbbbbbaaaa	ld.sb	LoaD, Signed Byte
10001ssmbbbbbaaaa	ld.sw	LoaD, Signed Wyde
10000ssmbbbbbaaaa	ld.ub	LoaD, Unsigned Byte
10000ssmbbbbbaaaa	ld.uw	LoaD, Unsigned Wyde
10001ssmbbbbbaaaa	ld.w	LoaD, Wyde
1010rrrrrrrrrrrr	ldi	LoaD Immediate
10011ssmbbbbbaaaa	lea	Load Effective Address
0111001bbbbbaaaa	lsl	Logical Shift Left
0111000bbbbbaaaa	lsr	Logical Shift Right
00000ttbbbbbaaaa	mov	MOVe
00001ttbbbbbaaaa	mov.not	MOVe NOT
011000100000aaaa	neg	NEGate
0000000000000000	nop	No OPeration
001100111111aaaa	not	NOT
00100ttbbbbbaaaa	or	OR
00101ttbbbbbaaaa	or.not	OR NOT
0111101bbbbbaaaa	rol	Rotate Left
0111100bbbbbaaaa	ror	Rotate Right
01100ttbbbbbaaaa	rsub	Reverse SUBtract
01101ttbbbbbaaaa	rsub.snb	Reverse SUBtract, Skip No Borrow
1111110000100000	rti	ReTurn from Interrupt
1111100000100000	rts	ReTurn from Subroutine

```
Opcode                 Mnemonic   Name
-------------------------------------------------------------------
1101010000000000          skip    SKIP
1101010000000000         skip.a   SKIP Always
110101010110aaaa        skip.abm  SKIP Any Byte Minus
110101010010aaaa        skip.abz  SKIP Any Byte Zero
110101010101aaaa        skip.awm  SKIP Any Wyde Minus
110101010001aaaa        skip.awz  SKIP Any Wyde Zero
1101111bbbbbaaaa         skip.bc  SKIP Bit Clear
1101011bbbbbaaaa         skip.bs  SKIP Bit Set
11010100bbbbaaaa         skip.eq  SKIP Equal
1101111010111aaaa        skip.fc  SKIP Flag Clear
1101010101111aaaa        skip.fs  SKIP Flag Set
11011010bbbbaaaa         skip.ge  SKIP Greater than or Equal
11011011bbbbaaaa         skip.gt  SKIP Greater Than
110111010100aaaa        skip.gtz  SKIP Greater than Zero
11011001bbbbaaaa         skip.hi  SKIP Higher
11011000bbbbaaaa         skip.hs  SKIP Higher or Same
11010011bbbbaaaa         skip.le  SKIP Less than or Equal
110101010100aaaa        skip.lez  SKIP Less than or Equal Zero
11010000bbbbaaaa         skip.lo  SKIP Lower
11010001bbbbaaaa         skip.ls  SKIP Lower or Same
11010010bbbbaaaa         skip.lt  SKIP Less Than
110101111111aaaa         skip.mi  SKIP Minus
1101110000000000          skip.n  SKIP Never
110111010110aaaa        skip.nbm  SKIP No Byte Minus
110111010010aaaa        skip.nbz  SKIP No Byte Zero
11011100bbbbaaaa         skip.ne  SKIP Not Equal
110111010101aaaa        skip.nwm  SKIP No Wyde Minus
110111010001aaaa        skip.nwz  SKIP No Wyde Zero
110111010000aaaa         skip.nz  SKIP NonZero
110111111111aaaa         skip.pl  SKIP Plus
110101010000aaaa          skip.z  SKIP Zero
11110nnnmmmmmmmm       spam.and   Skip Propagate Against Mask, AND mode
11110nnnmmmmmmmm       spam.xor   Skip Propagate Against Mask, XOR mode
10010ssmbbbbaaaa             st   STore
10010ssmbbbbaaaa           st.b   STore, Byte
10010ssmbbbbaaaa           st.q   STore, Quad
10010ssmbbbbaaaa           st.w   STore, Wyde
01010ttbbbbbaaaa            sub   SUBtract
01011ttbbbbbaaaa         sub.snb  SUBtract, Skip No Borrow
00110ttbbbbbaaaa            xor   XOR
00111ttbbbbbaaaa         xor.not  XOR NOT

       DIRECTIVE         .verify  .VERIFY
       DIRECTIVE           align  ALIGN
       DIRECTIVE             end  END
       DIRECTIVE             equ  EQUate
       DIRECTIVE             org  ORiGin
       DIRECTIVE            dc.b  Define Constant, Byte
       DIRECTIVE            dc.q  Define Constant, Quad
       DIRECTIVE            dc.s  Define Constant, String
       DIRECTIVE            dc.w  Define Constant, Wyde
       DIRECTIVE            dc.z  Define Constant, Zero terminated string
```

9 Instruction Encoding

9.1 Logical and Move Group

```
15:12  11     10:9     8:4   3:0      Operation
-------------------------------------------------------------------------
0000   NOTB   OPB_CTL  OPB    RA       MOV        RA  =          {NOT} OPB
0001   NOTB   OPB_CTL  OPB    RA       AND        RA  = RA  AND {NOT} OPB
0010   NOTB   OPB_CTL  OPB    RA       OR         RA  = RA  OR  {NOT} OPB
0011   NOTB   OPB_CTL  OPB    RA       XOR        RA  = RA  XOR {NOT} OPB
```

Abbreviations:

```
RA      : 4 bit register field (source 1 & destination)

OPB     : 5 bit immediate/4 bit register field (source 2)

OPB_CTL : select OPB type
           00: register
           01: 5 bit sign extended immediate
           10: 2^N immediate
           11: 2^N - 1 immediate

NOTB    : if set, use NOT OPB
```

9.2 Arithmetic Group

```
15:12  11     10:9     8:4   3:0      Operation
-------------------------------------------------------------------------
0100   CSKIP  OPB_CTL  OPB    RA       ADD        RA  = RA  + OPB
0101   CSKIP  OPB_CTL  OPB    RA       SUB        RA  = RA  - OPB
0110   CSKIP  OPB_CTL  OPB    RA       RSUB       RA  = OPB - RA

15:12  11:10   9       8:4   3:0      Operation
-------------------------------------------------------------------------
0111    00     0        N     RA       LSR        RA  = RA LSR  N
0111    00     1        N     RA       LSL        RA  = RA LSL  N
0111    01     0        N     RA       ASR        RA  = RA ASR  N
0111    01     1        N     RA       FLIP(*)    RA  = RA FLIP N
0111    10     0        N     RA       ROR        RA  = RA ROR  N
0111    10     1        N     RA       ROL        RA  = RA ROL  N

15:12  11:10   9:8     7:4   3:0      Operation
-------------------------------------------------------------------------
0111    11     00       RB    RA       reserved
0111    11     01       RB    RA       reserved
0111    11     10       RB    RA       FF1        RA  = Find First 1 RB
0111    11     11       RB    RA       CNT1       RA  = Count 1's    RB
```

Abbreviations:

```
RA      : 4 bit register field (source 1 & destination)
RB      : 4 bit register field (source 2)

OPB     : 5 bit immediate/4 bit register field (source 2)

OPB_CTL : select OPB type
           00: register
           01: 5 bit sign extended immediate
           10: 2^N immediate
           11: 2^N - 1 immediate

CSKIP   : if set, skip the next instruction if NO carry/borrow occurred
          if cleared, normal execution of following instruction

N       : 5 bit field
           - bit count for shift and rotate instructions

           - bit swap enable for FLIP
              N(4) : swap even/odd wydes
              N(3) : swap even/odd bytes
              N(2) : swap even/odd nybbles
              N(1) : swap even/odd bit pairs
              N(0) : swap even/odd bits
             e.g.
              11000 : byte reverse register ( swap wydes & bytes )
              11111 : bit reverse register  ( swap everything )
              00111 : bit reverse all bytes in register
```

[3]universal bit reverse per H. Warren, "Hacker's Delight", 2003, page 102, FLIP instruction credited to Guy Steele

9.3 Memory Group

```
15:12  11   10:9   8    7:4  3:0      Operation
-------------------------------------------------------------------------------
1000  SIGN  SIZE  OFS    RB   RA      LD        RA  = {extend} [ RB {+ IMM } ]
1001    0   SIZE  OFS    RB   RA      ST        [ RB {+ IMM } ] = {trunc}  RA

1001    1   SIZE  OFS    RB   RA      LEA       RA  = RB { + IMM}


15:12                       11:0      Operation
-------------------------------------------------------------------------------
1010                        EA12      LDI       LoaD Immediate  IMM = [EA12*4 + PC & $FFFF_FFFC]


15:12                       11:0      Operation
-------------------------------------------------------------------------------
1011                        I12       IMM12     IMM = sign extend I12
```

Abbreviations:

```
RA     : 4 bit register field (destination)
RB     : 4 bit register field (address)

SIGN   : when set, sign extend memory operand on load

SIZE   : load/store operand size
           00  quad (32 bit)  stack offset addressing
           01  quad (32 bit)
           10  wyde (16 bit)
           11  byte (8 bit)

OFS    : 0 = no offset,  1 = use IMM register   for normal addressing modes
         0 = fp          1 = sp                 for stack offset addressing mode

I12    : signed 12 bit immediate for IMM12

EA12   : unsigned 12 bit quad offset for LDI
```

9.4 Control Group

```
15:12  11      10:9   8    7:4  3:0     Operation
-------------------------------------------------------------------------------
1100   ?        ?     ?      ?    ?     CP            coprocessor

15:12                          11:0     Operation
-------------------------------------------------------------------------------
1101                            SCC     SKIP.CC       if (cond) skip next instruction

15:12  11      10     9          8:0     Operation
-------------------------------------------------------------------------------
1110   LBRA    PSHPC  DSLOT      I9     {L}BRA/{L}BSR  { push PC };
                                                      PC = PC + ( {IMM << 10} + I.9) * 2
15:12  11      10:8               7:0     Operation
-------------------------------------------------------------------------------
1111   0       TRUNC             MASK     SPAM.AND|XOR  skip propagate

15:12  11  10      9    8  7:4  3:0     Operation
-------------------------------------------------------------------------------
1111   1  PSHPC  DSLOT  0  0000  RA     JMP/JSR       { push PC }; PC = RA

1111   1  PSHPC  DSLOT  0  0001  RA     RBRA/RBSR     { push PC }; PC = PC + RA

1111   1  POPSR  DSLOT  0  0010  0000   RTS/RTI       pop PC; { pop SR }

15:12  11  10     9      8      7:0     Operation
-------------------------------------------------------------------------------
1111   1   1      DSLOT  1       TR     TRAP          push PC,SR; PC = vector(TR)
```

Abbreviations:

```
RA     : 4 bit register field (source 1)

DSLOT  : enables delay slot execution

PSHPC  : push PC
POPSR  : pop SR  ( used to turn an RTS into an RTI )

LBRA   : enables long branch: augments IMM9 offset with IMM[11:0] << 10

I9     : signed nine bit instruction offset for BRA
         ( for LBRA, I9 is the low 9 bits of a 21 | 31 bit signed offset )

SCC    : skip condition field, see table in next section

TRAP   : 8 bit trap vector

MASK   : skip propagate mask
           1 = skip enable
          D7 = next instruction .. D0 = eighth instruction

TRUNC  : skip propagate truncate field
           7    : AND mode
           6..0 : XOR mode, truncate skip mask by N instructions
```

9.5 Skip Condition Field

SCC : skip condition

11	10:8	7:4	3:0			
0	000	RB	RA	skip.lo	lower	unsigned, RA < RB
1	000	RB	RA	skip.hs	higher or same	unsigned, RA >= RB
0	001	RB	RA	skip.ls	lower or same	unsigned, RA <= RB
1	001	RB	RA	skip.hi	higher	unsigned, RA > RB
0	010	RB	RA	skip.lt	less than	signed, RA < RB
1	010	RB	RA	skip.ge	greater than or equal	signed, RA >= RB
0	011	RB	RA	skip.le	less than or equal	signed, RA <= RB
1	011	RB	RA	skip.gt	greater than	signed, RA > RB
0	100	RB	RA	skip.eq	equal	RA == RB
1	100	RB	RA	skip.ne	not equal	RA != RB
0	101	0000	RA	skip.z	zero	RA == 0
1	101	0000	RA	skip.nz	non-zero	RA != 0
0	101	0001	RA	skip.awz	any wyde zero	
1	101	0001	RA	skip.nwz	no wyde zero	
0	101	0010	RA	skip.abz	any byte zero	
1	101	0010	RA	skip.nbz	no byte zero	
0	101	0011	RA		reserved	
1	101	0011	RA		reserved	
0	101	0100	RA	skip.lez	less than or equal zero	RA <= 0
1	101	0100	RA	skip.gtz	greater than zero	RA > 0
0	101	0101	RA	skip.awm	any wyde minus	
1	101	0101	RA	skip.nwm	no wyde minus	
0	101	0110	RA	skip.abm	any byte minus	
1	101	0110	RA	skip.nbm	no byte minus	
0	101	0111	N	skip.fs	flag N set	test input flag bits
1	101	0111	N	skip.fc	flag N clear	
0	101	1ccc	N	skip.cpt	coprocessor N true	
1	101	1ccc	N	skip.cpf	coprocessor N false	

(continued on next page)

```
  Aliased skips:

       skip plus/minus aliases to "skip.bc/bs rn, #31" :

         11:0
       --------------------------------------------------------------------------------
         11111111aaaa         skip.pl    plus                    RA >= 0
         01111111aaaa         skip.mi    minus                   RA <  0

       skip always/never aliases to "skip.eq/ne r0,r0" :

         11:0
       --------------------------------------------------------------------------------
         010000000000         skip.a     always
         110000000000         skip.n     never


  Skip on bit:

         11  10:9  8:4  3:0
       --------------------------------------------------------------------------------
          0  11     B    RA   skip.bs    bit RA.B set
          1  11     B    RA   skip.bc    bit RA.B clear
```

Abbreviations:

```
RA     : 4 bit register field (source 1)
RB     : 4 bit register field (source 2)

N      : 4 bit input flag or coprocessor number

B      : bit number (for "skip on bit")

ccc    : coprocessor skip type
```